```
rt_1():
  rt_2()
```

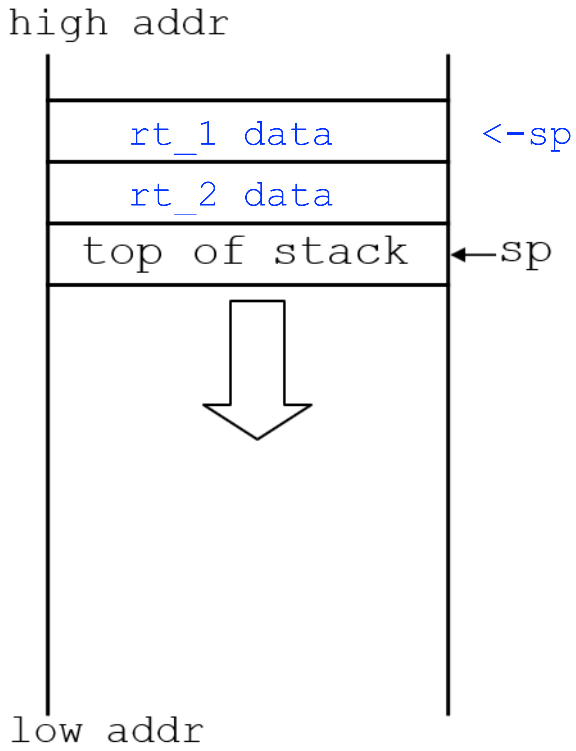
What if the callee needs to use more registers than allocated to argument and return values?

- Use a stack: a last-in-first-out queue
- One of the general registers, `sp`, is used to address the stack
- "grows" from high address to low address
- push: add data onto the stack, data on stack at new `sp`

```
sp = sp - 4
```

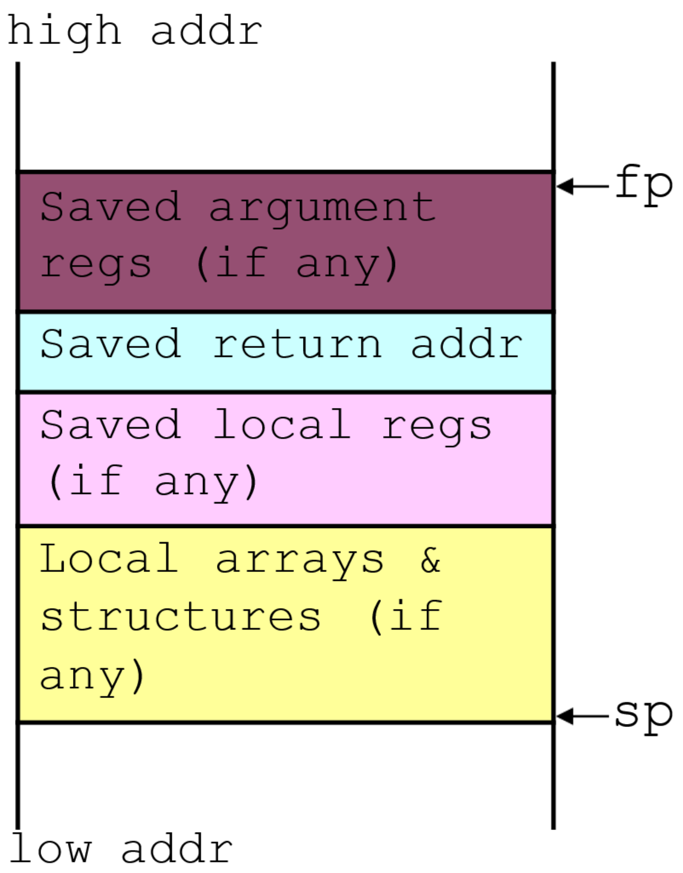
- pop: remove data from the stack, data from stack at `sp`

```
sp = sp + 4
```



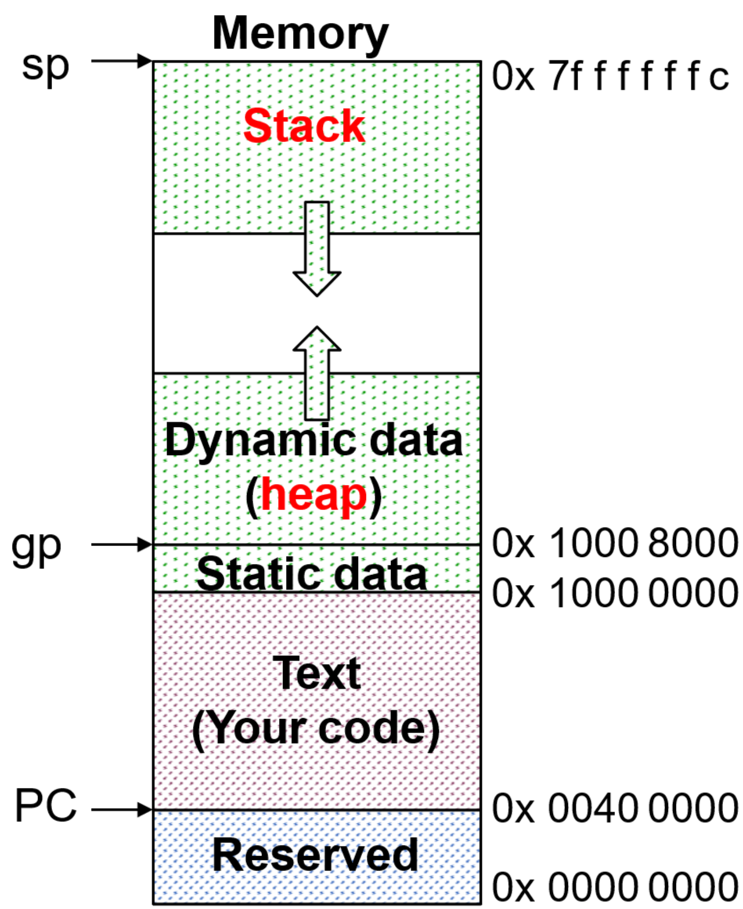
Allocating Space on the Stack

- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
- The frame pointer (`fp`) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
- `fp` is initialized using `sp` on a call and `sp` is restored using `fp` on a return



Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
- Allocate space on the heap with `malloc()` and free it with `free()` in C



EX-3: Compiling a C Leaf Procedure

Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

```
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Solution:

EX-3: Compiling a C Leaf Procedure

Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

```
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Q: what if we swap two sw instions?

```
sw t0, 4(sp)
sw t1, 0(sp)
…
lw t0, 4(sp)
lw t1, 0(sp)
```

Discussion: above lw order is not important; we just care about the relative address over sp

Solution:

Suppose g, h, i, and j are in a0, a1, a2, a3

```
leaf_ex:  addi   sp, sp, -8   # make stack room
          sw     t1, 4(sp)    # save t1 on stack
          sw     t0, 0(sp)    # save t0 on stack
          add    t0, a0, a1
          add    t1, a2, a3
          sub    s0, t0, t1
          lw     t0, 0(sp)    # restore t0
          lw     t1, 4(sp)    # restore t1
          addi   sp, sp, 8    # adjust stack ptr (pop two values from stack)
          jalr   zero, 0(ra)
```

Q: what if we lw t1, 8(sp)?
- we are accessing to data from other procedure
- in other words, we should carefully manipulate sp, so that stack is working as FILO

In this course we assume all based on 32 bit system.

Nested Procedures

- Nested Procedure: call other procedures
- What happens to return addresses with nested procedures? how to reuse ra?

```
int rt_1 (int i)
{
    if (i == 0) return 0;
    else return rt_2(i-1);
}
```

Nested procedures (cont.)

Problem of the left example:
ra is overwritten

What's the correct way?
– callee must save it into stack

```
caller: jal   rt_1
next:   . . .         <- ra

rt_1:   bne  a0, zero, to_2
        add  s0, zero, zero
        jalr zero, 0(ra)
to_2:   addi a0, a0, -1
        jal  ra, rt_2
        jalr zero, 0(ra) <- ra

rt_2:   . . .
```

Q: can we store ra to other registers, e.g. a5?
– Yes, as long as ra can be restored.

- On the call to `rt_1`, the return address (next in the caller routine) gets stored in `ra`.

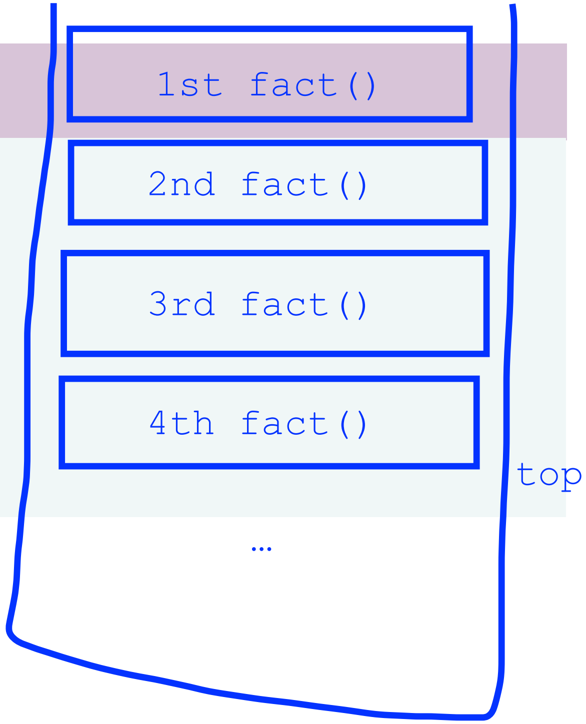
Question:

What happens to the value in `ra` (when `a0 != 0`) when `to_2` makes a call to `rt_2`?

Compiling a Recursive Procedure

A procedure for calculating factorial

```
int fact (int n)
{
    if (n < 1) return 1;
    else return (n * fact (n-1));
}
```



- A recursive procedure (one that calls itself!)

```
fact (0) = 1
fact (1) = 1 * 1 = 1
fact (2) = 2 * 1 * 1 = 2
fact (3) = 3 * 2 * 1 * 1 = 6
fact (4) = 4 * 3 * 2 * 1 * 1 = 24
. . .
```

- Assume n is passed in `a0`; result returned in `s0`

Compiling a Recursive Procedure (cont.)

Q: when we return from "jal ra, fact", where are we?
— on the line of "bk_f"

```
fact: addi  sp, sp, -8      # adjust stack pointer
      sw    ra, 4(sp)       # save return address
      sw    a0, 0(sp)       # save argument n
      slti  t0, a0, 1       # test for n < 1
      beq   t0, zero, L1    # if n >= 1, go to L1
      addi  s0, zero, 1     # else return 1 in s0
      addi  sp, sp, 8       # adjust stack pointer
      jalr  zero, 0(ra)     # return to caller
L1:   addi  a0, a0, -1      # n >= 1, so decrement n
      jal   ra, fact        # call fact with (n-1)
                            # this is where fact returns
bk_f: lw    a0, 0(sp)       # restore argument n
      lw    ra, 4(sp)       # restore return address
      addi  sp, sp, 8       # adjust stack pointer
      mul   s0, a0, s0      # s0 = n * fact(n-1)
      jalr  zero, 0(ra)     # return to caller
```

Note: `bk_f` is carried out when `fact` is returned.

Question:

Why we don't load `ra, a0` back to registers?